Cell morphing: from array programs to array-free Horn clauses

David Monniaux, Laure Gonnord

Cell morphing: from array programs to array-free Horn clauses*

David Monniaux
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble
CNRS, VERIMAG, F-38000 Grenoble, France

Laure Gonnord
LIP, Univ. Lyon-1, France

June 27, 2016

Abstract

Automatically verifying safety properties of programs is hard. Many approaches exist for verifying programs operating on Boolean and integer values (e.g. abstract interpretation, counterexample-guided abstraction refinement using interpolants), but transposing them to array properties has been fraught with difficulties. Our work addresses that issue with a powerful and flexible abstraction that morphes concrete array cells into a finite set of abstract ones. This abstraction is parametric both in precision and in the back-end analysis used.

From our programs with arrays, we generate nonlinear Horn clauses over scalar variables only, in a common format with clear and unambiguous logical semantics, for which there exist several solvers. We thus avoid the use of solvers operating over arrays, which are still very immature.

Experiments with our prototype VAPHOR show that this approach can prove automatically and without user annotations the functional correctness of several classical examples, including *selection sort*, *bubble sort*, *insertion sort*, as well as examples from literature on array analysis.

1 Introduction

In this article, we consider programs operating over arrays, or, more generally, *maps* from an index type to a value type (in the following, we shall use "array" and "map" interchangeably). Such programs contain read (e.g. $v := a[i]$) and write ($a[i] := v$) operations over arrays, as well as "scalar" operations.[1] We wish to fully automatically verify properties on such programs; e.g. that a sorting algorithm outputs a sorted permutation of its input.

Universally Quantified Properties Very often, desirable properties over arrays are universally quantified; e.g. sortedness may be expressed as $\forall k_1, k_2\ k_1 < k_2 \implies a[k_1] \leq a[k_2]$. However, formulas with universal quantification and linear arithmetic over integers and at least one predicate symbol (a predicate being a function to the Booleans) form an undecidable class [20], of which some decidable subclasses have however been identified [8]. There is therefore no general algorithm for checking that such invariants hold, let alone inferring them. Yet, there have been several approaches proposed to infer such invariants (see Sec. 7).

We here propose a method for inferring such universally quantified invariants, given a specification on the output of the program. This being undecidable, this approach may fail to terminate in the general case, or may return "unknown". Experiments however show that our approach can successfully and automatically verify nontrivial properties (e.g. the output from selection sort is sorted and is a permutation of the input).

Our key insight is that if there is a proof of safety of an array-manipulating program, it is likely that there exists a proof that can be expressed with simple steps over properties relating only a small number N of (parametric) array cells, called "distinguished cells" . For instance, all the sorting algorithms we tried can be proved correct with $N = 2$, and simple array manipulations (copying, reversing...) with $N = 1$.

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR"

[1] In the following, we shall lump as "scalar" operations all operations not involving the array under consideration, e.g. $i := i + 1$. Any data types (integers, strings etc.) are supported if supported by the back-end solver.

Horn Clauses We convert the verification problem to Horn clauses, a common format for program verification problems [35] supported by a number of tools. Usual conversions [18] map variables and operations from the program to variables of the same type and the same operations in the Horn clause problem:[2] an integer is mapped to an integer, an array to an array, etc. If arrays are not supported by the back-end analysis, they may be abstracted away (reads replaced by nondeterministic choices, writes discarded) at the expense of precision. In contrast, our approach abstracts programs much less violently, with tunable precision, even though the result is still a Horn clause problem without arrays. Section 3 explains how many properties (e.g. initialization) can be proved using one "distinguished cell" ($N = 1$), Section 4 explains how properties such as sortedness can be proved using two cells; completely discarding arrays corresponds to using zero of them.

An interesting characteristic of the Horn clauses we produce is that they are nonlinear,[3], even though a straightforward translation of the semantics of a control-flow graph into Hoare triples expressed as clauses yields a linear system (whose unfoldings correspond to abstract execution traces). If a final property to prove (e.g. "all values are 0") queries one cell position, this query may morph, by the backward unfolding of the clauses, into a tree of queries at other locations, in contrast to some earlier approaches [31].

We illustrate this approach with automated proofs of several examples from the literature: we apply Section 3, 4 or 5 to obtain a system of Horn clauses without arrays. This system is then fed to the Z3, Eldarica or Spacer solver, which produces a model of this system, meaning that the postcondition (e.g. sortedness or multiset of the output equal to that of the input) truly holds.[4]

Previous approaches [31] using "distinguished cells" amounted (even though not described as such) to linear Horn rules; on contrast, our abstract semantics uses non-linear Horn rules, which leads to higher precision (Sec. 7.1).

Contributions Our main contribution is a system of rules for transforming the atomic program statements in a program operating over arrays or maps, as well as (optionally) the universally quantified postcondition to prove, into a system of non-linear Horn clauses over scalar variables only. The precision of this transformation is tunable using a Galois connection parameterized by the number of "distinguished cells"; e.g. properties such as sortedness need two distinguished cells (Section 4) while simpler properties need only one (Section 3). Statements operating over non-arrays variables are mapped (almost) identically to their concrete semantics. This system over-approximates the behavior of the program. A solution of that system can be mapped to inductive invariants over the original programs, including universal properties over arrays.

A second contribution, based on the first, is a system of rules that also keeps track of array/map contents (Sec. 5) as a multiset. This system is suitable for showing content properties, e.g. that the output of a sorting algorithm is a permutation of the input, even though the sequence of operations is not directly a sequence of swaps.

We implemented our approach and benchmarked it over several classical examples of array algorithms (Section 6), comparing it favorably to other tools.

2 Program Verification as solving Horn clauses

A classical approach to program analysis is to consider a program as a control-flow graph and to attach to each vertex p_i (control point) an *inductive invariant* I_i: a set of possible values $\mathbf{x}$ of the program variables (and memory stack and heap, as needed) so that i) the set associated to the initial control point p_{i_0} contains the possible initialization values S_{i_0} ii) for each edge $p_i \rightarrow_c p_j$ (c for *concrete*), the set I_j associated to the target control point p_j should include all the states reachable from the states in the set I_i associated to the source control point p_i according to the transition relation $\tau_{i,j}$ of the edge.

[2] With the exception of pointers and references, which need special handling and may be internally converted to array accesses.

[3] A nonlinear clause is of the form $P_1(\dots) \wedge P_2(\dots) \wedge \dots \wedge P_n(\dots) \wedge \textit{arithmetic condition} \implies Q(\dots)$, with several antecedent predicates $P_1, P_2, \dots$. Unfolding such rules yields a tree. In contrast a linear rule $P(\dots) \wedge \textit{arithmetic condition} \implies Q(\dots)$ has only one antecedent predicate, and unfolding a system of such rules yields a linear chain.

[4] Z3 and Eldarica can also occasionally directly solve Horn clauses over arrays; we also compare to that.

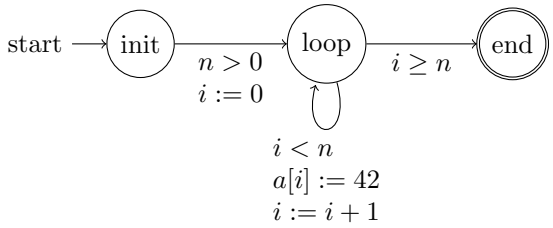

Figure 1: Compact control-flow graph for Program 1

Inductiveness is thus defined by *Horn clauses*[5]:

$$\forall \mathbf{x},\ S_{i_0}(\mathbf{x}) \implies I_{i_0}(\mathbf{x}) \tag{1}$$

$$\forall \mathbf{x}, \mathbf{x}',\ I_i(\mathbf{x}) \wedge \tau_{i,j}(\mathbf{x}, \mathbf{x}') \implies I_j(\mathbf{x}') \tag{2}$$

For proving safety properties, in addition to inductiveness, one requires that error locations $p_{e_1}, \ldots, p_{e_n}$ are proved to be unreachable (the associated set of states is empty): this amounts to Horn clauses implying false: $\forall \mathbf{x},\ I_{e_i}(\mathbf{x}) \Rightarrow \bot$.

Various tools can solve such systems of Horn clauses, that is, can synthesize suitable predicates I_i, which constitute *inductive invariants*. In this article, we tried Z3[6] with the PDR fixed point solver [22], Z3 with the Spacer solver [24, 25],[7] and Eldarica[36].[8] Since program verification is undecidable, such tools, in general, may fail to terminate, or may return "unknown".

For the sake of simplicity, we shall consider, in this article, that all integer variables in programs are mathematical integers ($\mathbb{Z}$) as opposed to machine integers[9] and that arrays are infinite. Again, it is easy to modify our semantics to include systematic array bound checks, jumps to error conditions, etc.

In examples, instead of writing I_{stmt} for the name of the predicate (inductive invariant) at statement *stmt*, we shall write *stmt* directly, for readability's sake: thus we write e.g. *loop* for a predicate at the head of a loop. Furthermore, for readability, we shall sometimes coalesce several successive statements into one.

Example 1 (Motivating example)**.** *Consider the program:*

```
void array_fill1(int n, int a[n]) {
  for(int i=0; i<n; i++) a[i]=42;
  /* assert ∀0 ≤ k < n, a[k] = 42 */
}
```

We would like to prove that this program truly fills array a[] *with value 42. The flat encoding into Horn clauses assigns a predicate (set of states) to each of the control nodes (Fig. 1), and turns each transition into a Horn rule with variables ranging in Array* (A, B)*, the type of arrays of* B *indexed by* A *[26, Ch. 7]:*

$$\forall n \in \mathbb{Z}\ \forall a \in \mathit{Array}(\mathbb{Z}, \mathbb{Z})\ \ n > 0 \implies \mathit{loop}(n, 0, a) \tag{3}$$

$$\forall n, i \in \mathbb{Z}\ \forall a \in \mathit{Array}(\mathbb{Z}, \mathbb{Z})\ \ i < n \wedge \mathit{loop}(n, i, a) \implies \mathit{loop}(n, i+1, \mathit{store}(a, i, 42)) \tag{4}$$

$$\forall n, i \in \mathbb{Z}\ \forall a \in \mathit{Array}(\mathbb{Z}, \mathbb{Z})\ \ i \geq n \wedge \mathit{loop}(n, i, a) \implies \mathit{end}(n, i, a) \tag{5}$$

$$\forall x, n, i \in \mathbb{Z}\ \forall a \in \mathit{Array}(\mathbb{Z}, \mathbb{Z})\ \ 0 \leq x < n \wedge \mathit{end}(n, i, a) \implies \mathit{select}(a, x) = 42 \tag{6}$$

where $\mathit{store}(a, i, v)$ *is array* a *where the value at index* i *has been replaced by* v *and* $\mathit{select}(a, x)$ *denotes* $a[x]$*.*

None of the tools we have tried (Z3, Spacer, Eldarica) has been able to solve this system, presumably because they cannot infer universally quantified invariants over arrays.[10] Indeed, here the loop

[5] Classically, we denote the sets using predicates: $I_{i_0}(\mathbf{x})$ means $\mathbf{x} \in I_{i_0}$

[6] `https://github.com/Z3Prover` hash 7f6ef0b6c0813f2e9e8f993d45722c0e5b99e152; due to various problems we preferred not to use results from later versions.

[7] `https://bitbucket.org/spacer/code` hash 7e1f9af01b796750d9097b331bb66b752ea0ee3c

[8] `https://github.com/uuverifiers/eldarica/releases/tag/v1.1-rc`

[9] A classical approach is to add overflow checks to the intermediate representation of programs in order to be able to express their semantics with mathematical integers even though they operate over machine integers.

[10] Some of these tools can however infer some simpler array invariants.

invariant needed is

$$0 \leq i \leq n \wedge (\forall k\ 0 \leq k < i \implies a[k] = 42) \tag{7}$$

While $0 \leq i \leq n$ is inferred by a variety of approaches, the rest is tougher. We shall also see (Ex. 6) that a slight alteration of this example also prevents some earlier abstraction approaches from checking the desired property.

Most software model checkers attempt constructing invariants from *Craig interpolants* obtained from refutations [9] of the accessibility of error states in local [22] or global [30] unfoldings of the problem. However, interpolation over array properties is difficult, especially since the goal is not to provide any interpolant, but interpolants that generalize well to invariants [1, 2]. This article instead introduces a way to derive universally quantified invariants from the analysis of a system of Horn clauses on scalar variables (without array variables).

3 Getting Did of the Arrays

To use the power of Horn solvers, we soundly abstract problems with arrays to problems without arrays. In the Horn clauses for Ex. 1, we attached to each program point p_ℓ a predicate I_ℓ over $\mathbb{Z}\times\mathbb{Z}\times \mathit{Array}\,(\mathbb{Z},\mathbb{Z})$ when the program variables are two integers i, n and one integer-value, integer-indexed array a. [11] In any solution of the system of clauses, if the valuation (i, n, a) is reachable at program point p_ℓ, then $I_\ell(i, n, a)$ holds. Instead, in the case of Ex. 1, we will consider a predicate $I^\sharp_\ell$ over $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ (the array *key* $\rightarrow$ *value* has been replaced by a pair $(\mathit{key}, \mathit{value})$) such that $I^\sharp_\ell(i, n, k, a_k)$ [12] holds for each reachable state (i, n, a) satisfying $a[k] = a_k$. This is the same Galois connection [11] as some earlier works [31] [13, Sec. 2.1]; yet, as we shall see, our abstract transformers are more precise.

Definition 1. The "one distinguished cell" *abstraction* of $I \subseteq \chi \times \mathit{Array}\,(\iota, \beta)$ is $\alpha\,(I) = \{(\mathbf{x}, i, a[i]) \mid \mathbf{x} \in \chi, i \in \iota\}$. The *concretization* of $I^\sharp \subseteq \chi \times (\iota \times \beta)$ is $\gamma\left(I^\sharp\right) = \{(\mathbf{x}, a) \mid \forall i \in \iota\ (\mathbf{x}, i, a[i]) \in I^\sharp\}$.

Theorem 1. $\mathcal{P}\,(\chi \times \mathit{Array}\,(\iota, \beta)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}\,(\chi \times (\iota \times \beta))$ *is a Galois connection.*

To provide the abstract transformers, we will suppose in the sequel that any statement in the program (*control flow graph*) will be: i) either an array read to a fresh variable, `v=a[i];` in C syntax, $v := a[i]$ in pseudo-code; the variables of the program are $(\mathbf{x}, i, v)$ where $\mathbf{x}$ is a vector of arbitrarily many variables; ii) either an array write, `a[i]=v;` (where v and i are variables) in C syntax, $a[i] := v$ in pseudo-code; the variables of the program are $(\mathbf{x}, i, v)$ before and after the statement; iii) or a scalar operation, including assignments and guards over scalar variables. More complex statements can be transformed to a sequence of such statements, by introducing temporary variables if needed: for instance, $a[i] := a[j]$ is transformed into $\mathit{temp} := a[j];\ a[i] := \mathit{temp}$.

Definition 2 (Read statement)**.** Let v be a variable of type β, i be a variable of type ι, and a be an array of values of type β with an index of type ι. Let $\mathbf{x}$ be the other program variables, taken in χ. The concrete "next state" relation for the read statement `v=a[i];` between locations p_1 and p_2 is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}, i, a[i], a)$.

Its forward abstract semantics is encoded into two Horn clauses:

$$\begin{aligned} &\forall \mathbf{x} \in \chi\ \forall i \in \iota\ \forall v, a_i \in \beta\ \forall k \in \iota\ \forall a_k \in \beta \\ k \neq i \wedge I^\sharp_1\big((\mathbf{x}, i, v), (k, a_k)\big) \wedge I^\sharp_1\big((\mathbf{x}, i, v), (i, a_i)\big) &\implies I^\sharp_2\big((\mathbf{x}, i, a_i), (k, a_k)\big) \end{aligned} \tag{8}$$

$$\begin{aligned} &\forall \mathbf{x} \in \chi\ \forall i \in \iota\ \forall v, a_i \in \beta\ \forall k \in \iota\ \forall a_k \in \beta \\ I^\sharp_1\big((\mathbf{x}, i, v), (i, a_i)\big) &\implies I^\sharp_2\big((\mathbf{x}, i, a_i), (i, a_i)\big) \end{aligned} \tag{9}$$

The tuple (k, a_k) now represents a "distinguished cell". While rule 9 is straightforward (a_i is assigned to the variable v), the nonlinear rule 8 may be more difficult to comprehend. The intuition is that, to have both $a_i = a[i]$ and $a_k = a[k]$ at the read instruction with a given valuation $(\mathbf{x}, i)$ of the other variables, both $a_i = a[i]$ and $a_k = a[k]$ had to be reachable with the same valuation.

[11] For instance, $I_{loop} = \mathit{loop}(n, i, a)$, $I_{end} = \mathit{end}(n, i, a)$.

[12] also denoted by $I^\sharp_\ell((i, n), (k, a_k))$ for sake of readability.

Remark 1. *We use two separate rules for $k = i$ and $k \neq i$ for better precision. A single rule $I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i)\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (k, a_k)\big)$ would not enforce that if $i = k$ then $a_i = a_k$ in the consequent.*

From now on, we shall omit all universal quantifiers inside rules, for readability.

Definition 3 (Write statement)**.** The concrete "next state" relation for the write statement `a[i]=v;` is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}, i, v, store(a, i, v))$. Its forward abstract semantics is encoded into two Horn clauses, depending on whether the distinguished cell is i or not:

$$I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \wedge i \neq k \implies I_2^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \tag{10}$$

$$I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i)\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (i, v)\big) \tag{11}$$

Example 2 (Ex. 1, cont.)**.** *The $a[i] := 42$ statement of Ex. 1 is translated into (the loop control point is divided into loop/write/incr, all predicates of arity 4):*

$$i \neq k \wedge write(n, i, k, a_k) \implies incr(n, i, k, a_k) \tag{12}$$

$$write(n, i, i, a_i) \implies incr(n, i, i, 42) \tag{13}$$

Definition 4 (Initialization)**.** The creation of an array variable with nondeterministically chosen initial content is abstracted by $I_1^\sharp(\mathbf{x}) \implies I_2^\sharp(\mathbf{x}, k, a_k)$.

Definition 5 (Scalar statements)**.** With the same notations as above, we consider a statement (or sequence thereof) operating only on scalar variables: $\mathbf{x} \rightarrow_s \mathbf{x}'$ if it is possible to obtain scalar values $\mathbf{x}'$ after executing the statement on scalar values $\mathbf{x}$. The concrete "next state" relation for that statement is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}', i, v, a)$. Its forward abstract semantics is encoded into:

$$I_1^\sharp(\mathbf{x}, k, a_k) \wedge \mathbf{x} \rightarrow_s \mathbf{x}' \implies I_2^\sharp(\mathbf{x}', k, a_k) \tag{14}$$

Example 3. *A test $x \neq y$ gets abstracted as*

$$I_1^\sharp(x, y, k, a_k) \wedge x \neq y \implies I_2^\sharp(x, y, k, a_k) \tag{15}$$

Definition 6. The scalar operation $kill(v_1, \ldots, v_n)$ removes variables $v_1, \ldots, v_n$: $(\mathbf{x}, v_1, \ldots, v_n) \rightarrow \mathbf{x}$. We shall apply it to get rid of dead variables, sometimes, for the sake of brevity, without explicit note, by coalescing it with other operations.

Our Horn rules are of the form $\forall \mathbf{y}\ I_1^\sharp(\mathbf{f_1}(\mathbf{y})) \wedge \cdots \wedge I_1^\sharp(\mathbf{f_m}(\mathbf{y})) \wedge P(\mathbf{y}) \implies I_2^\sharp(\mathbf{g}(\mathbf{y}))$ ($\mathbf{y}$ is a vector of variables, $\mathbf{f_1}, \ldots, \mathbf{f_m}$ vectors of terms depending on $\mathbf{y}$, P an arithmetic predicate over $\mathbf{y}$). In other words, they impose in $I_2^\sharp$ the presence of $\mathbf{g}(\mathbf{y})$ as soon as certain $\mathbf{f_1}(\mathbf{y}), \ldots, \mathbf{f_m}(\mathbf{y})$ are found in $I_1^\sharp$. Let $I_{2\_}^\sharp$ be the set of such imposed elements. This Horn rule is said to be *sound* if $\gamma(I_{2\_}^\sharp)$ includes all states $(\mathbf{x}', a')$ such that there exists $(\mathbf{x}, a)$ in $\gamma(I_1^\sharp)$ and $(\mathbf{x}, a) \rightarrow_c (\mathbf{x}', a')$.

Lemma 2. *The forward abstract semantics of the read statement (Def. 2), of the write statement (Def. 3), of array initialization (Def. 4), of the scalar statements (Def. 5) are sound w.r.t the Galois connection.*

Remark 2. *The scalar statements include "killing" dead variables (Def. 6). Note that, contrary to many other abstractions, in ours, removing some variables may cause irrecoverable loss of precision on other variables [31, Sec. 4.2]: if v is live, then one can represent $\forall k,\ a[k] = v$, which implies $\forall k_1, k_2\ a[k_1] = a[k_2]$ (constantness), but if v is discarded, the constantness of a is lost.*

Theorem 3. *If $I_1^\sharp, \ldots, I_m^\sharp$ are a solution of a system of Horn clauses sound in the above sense, then $\gamma(I_1^\sharp), \ldots, \gamma(I_m^\sharp)$ are inductive invariants w.r.t the concrete semantics $\rightarrow_c$.*

Definition 7 (Property conversion)**.** A property "at program point p_ℓ, for all $\mathbf{x} \in \chi$ and all $k \in \iota$, $\phi(\mathbf{x}, k, a[k])$ holds" (where ϕ is a formula, say over arithmetic) is converted into a Horn query $\forall \mathbf{x} \in \chi\ \forall k \in \iota\ I_\ell^\sharp(\mathbf{x}, k, a_k) \implies \phi(\mathbf{x}, k, a_k)$.

Our method for converting a scalar program into a system of Horn clauses over scalar variables is thus:

Algorithm 1 (Abstraction into Horn constraints)**.** Given the control-flow graph of the program:

1. To each control point p_ℓ, with vector of scalar variables $\mathbf{x}_\ell$, associate a predicate $I^\sharp_\ell(\mathbf{x}_\ell, k, a_k)$ in the Horn clause system (the vector of scalar variables may change from control point to control point).
2. For each transition of the program, generate Horn rules according to Def. 2, 3, 5 as applicable (an initialization node has no antecedents in its rule).
3. Generate Horn queries from desired properties according to Def. 7.

Example 4 (Ex. 1, continued)**.** *Let us now apply the Horn abstract semantics from Definitions 3 and 5 to Program 1; in this case,* $\beta = \mathbb{Z}$, $\iota = \{0, \ldots, n-1\}$ *(thus we always have* $0 \leq k < n$*)* , $\chi = \mathbb{Z}$. *After slight simplification, we get:*

$$0 \leq k < n \implies loop(n, 0, k, a_k) \tag{16}$$

$$0 \leq k < n \wedge i < n \wedge loop(n, i, k, a_k) \implies write(n, i, k, a_k) \tag{17}$$

$$0 \leq k < n \wedge i \neq k \wedge write(n, i, k, a_k) \implies incr(n, i, k, a_k) \tag{18}$$

$$write(n, i, i, a_i) \implies incr(n, i, i, 42) \tag{19}$$

$$0 \leq k < n \wedge incr(n, i, k, a_k) \implies loop(n, i+1, k, a_k) \tag{20}$$

$$0 \leq k < n \wedge i \geq n \wedge loop(n, i, k, a_k) \implies end(n, i, k, a_k) \tag{21}$$

Finally, we add the postcondition (using Def. 7):

$$0 \leq k < n \wedge end(n, i, k, a_k) \Rightarrow a_k = 42 \tag{22}$$

A solution to the resulting system of Horn clauses can be found by e.g. Z3.

Our approach can also be used to establish relationships between several arrays, or between the initial values in an array and the final values: arrays $a[i]$ and $b[j]$ can be abstracted by a quadruple (i, a_i, j, b_j).[13]

Example 5. *Consider the problem of finding the minimum of an array slice* $a[d \ldots h-1]$*, with value* $b = a[p]$*:*

```
void find_minimum(int n, int a[n], int d, int h){
  int p = d, b = a[d], i = d+1;
  while(i < h) {
    if (a[i] < b) {
      b = a[i];
      p = i;
    }
    i = i+1;
  }
}
```

Again, we encode the abstraction of the statements (Def. 2, 3, 5) as Horn clauses. We obtained a predicate $end(d, h, p, b, k, a_k)$ *constrained as follows:*

$$end(d, h, p, b, p, a_p) \implies b = a_p \tag{23}$$

$$d \leq k < h \wedge end(d, h, p, b, k, a_k) \implies b \leq a_k \tag{24}$$

Rule 23 imposes the postcondition $b = a[p]$*, Rule 24 imposes the postcondition* $\forall k\ d \leq k < h \implies b \leq a[k]$*. This example is again solved by* Z3.

[13] If one is sure that the only relation that matters between a and b are between cells of same index, then one can use triples (i, a_i, b_i).

Earlier approaches based on translation to programs [31], thus transition systems, are equivalent to translating into *linear* Horn clauses where $x_1, \ldots, x_p$ are the same in the antecedent and consequent:

$$I_1(\ldots, x_1, a_1, \ldots, x_p, a_p) \wedge \mathit{condition} \rightarrow I_2(\ldots, x_1, a'_1, \ldots, x_p, a'_p) \tag{25}$$

In contrast, in this article we use a much more powerful translation to nonlinear Horn clauses (Sec. 7.1) where $x''_1, \ldots, x''_p$ differ from $x_1, \ldots, x_p$:

$$I_1(\ldots, x_1, a_1, \ldots, x_p, a_p) \wedge \ldots \wedge I_1(\ldots, x''_1, a''_1, \ldots, a''_p, x''_p) \wedge \mathit{condition} \rightarrow I_2(\ldots, x_1, a'_1, \ldots, x_p, a'_p) \tag{26}$$

Example 6 (Motivating example, altered)**.** *The earlier work [31] could successfully analyze Ex 1. However a slight modification of the program prevents it from doing so:*

```
  int tab[n];
  for (int i=0; i<n; i++) tab[i]=42;
M: f = 0;
  for (int i=0; i<n; i++) { if (tab[i] != 42) f=1; }
  assert(f == 0);
```

For this particular example, the array tab *would be abstracted* all over the program *using a* fixed *number of cells* tab[x_1],...,tab[x_p]*, where* $x_1, \ldots, x_p$ *are symbolic constants.*

The second loop is then analyzed as though it were[14]*.*

```
  for (int i=0; i<n; i++) {
     r = random();
     if (i == x_1) r = a_1;
     ⋮
     if (i == x_p) r = a_p;
T:   if (r != 42) f=1;
  }
```

One easily sees that if $p < n$*, there must be a loop iteration where* $i \notin \{x_1, \ldots, x_p\}$ *and thus at location* T*,* r *takes any value and* f *may take value* 1*. The output of our translation process in this example is the same until the M control point, then*[15] *it is:*

$$\mathit{end}(n, i, k, a_k) \implies \mathit{loop2}(n, 0, x, k, a_k, 0) \tag{27}$$

$$\mathit{loop2}(n, i, x, k, a_k, f) \wedge i < n \implies \mathit{rd2}(n, i, x, k, a_k, f) \tag{28}$$

$$\mathit{rd2}(n, i, x, k, a_k, f) \wedge i \neq k \wedge \boldsymbol{rd2(n, i, x, i, a_i, f)} \implies \mathit{test2}(n, i, a_i, k, a_k, f) \tag{29}$$

$$\mathit{rd2}(n, i, x, i, a_i, f) \implies \mathit{test2}(n, i, a_i, i, a_i, f) \tag{30}$$

$$\mathit{test2}(n, i, x, k, a_k, f) \wedge x \neq 42 \implies \mathit{loop2}(n, i+1, k, a_k, 1) \tag{31}$$

$$\mathit{test2}(n, i, x, k, a_k, f) \wedge x = 42 \implies \mathit{loop2}(n, i+1, k, a_k, f) \tag{32}$$

$$\mathit{loop2}(n, i, x, k, a_k, f) \wedge i \geq n \implies \mathit{end2}(f) \tag{33}$$

$$\mathit{end2}(f) \Rightarrow f = 0 \quad \textit{(property to prove)} \tag{34}$$

This abstraction is precise enough to prove the desired property, thanks to the ***bold*** *antecedent (multiples cell indices occur within the same unfolding). Without this nonlinear rule (removing this antecedent yields a sound abstraction equivalent to [31]), the unfoldings of the system of rules all carry the same* k *(index of the distinguished cell) between predicates: the program is analyzed with respect to one single* $a[k]$*, with* k *symbolic, leading to insufficient precision.*

[14] It would still be possible to proceed by first analyzing the first loop, getting the scalar invariant $\mathsf{tab}_x = 42$ at location M, quantifying it universally as $\forall x$ tab[x], then analyzing the second loop. Such an approach would however fail if this program was itself included in an outer loop.

[15] The statement `if tab[i]!=42` is decomposed into `x:=a[i];if(x!=42)`.

No Restrictions on Domain Type and Relationships, and Matrices The kind of relationship that can be inferred between loop indices, array indices and array contents is limited only by the capabilities of the Horn solver. For instance, invariants of the form $\forall i\ i \equiv 0 \pmod 2 \implies a[i] = 0$ may be inferred if the Horn solver supports numeric invariants involving divisibility. Similarly, we have made no assumption regarding the nature of the indexing variable: we used integers because arrays indexed by an integer range are a very common kind of data structure, but really it can be any type supported by the Horn clause solver, e.g. rationals or strings. For instance, *matrices* (resp. *n-tensors*) are specified by having pairs of integers (resp. n-tuples) as indices.

4 Sortedness and Other N-ary Predicates

The Galois connection of Def. 1 expresses relations of the form $\forall k \in \iota\ \phi(\mathbf{x}, k, a[k])$ where $\mathbf{x}$ are variables from the program, a a map and k an index into the map a; in other words, relations between each array element individually and the rest of the variables. It cannot express properties such as sortedness, which link *two* array elements: $\forall k_1, k_2 \in \iota\ k_1 < k_2 \implies a[k_1] \leq a[k_2]$.

For such properties, we need two "distinguished cells", with indices k_1 and k_2. For efficiency, we break this symmetry between indices k_1 and k_2 by imposing $k_1 < k_2$ for some total order.

Definition 8. The abstraction with indices $k_1 < k_2$ is

$$\gamma_{2<}\left(I^\sharp\right) = \{(\mathbf{x}, a) \mid \forall k_1 < k_2 \in \iota\ (\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \in I^\sharp\} \tag{35}$$

$$\alpha_{2<}\left(I\right) = \{(\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \mid x \in \chi, k_1 \leq k_2 \in \iota\} \tag{36}$$

Theorem 4. *$\alpha_{2<}$ and $\gamma_{2<}$ form a Galois connection:*

$$\mathcal{P}\left(\chi \times \mathit{Array}\left(\iota, \beta\right)\right) \underset{\alpha_{2<}}{\overset{\gamma_{2<}}{\leftrightarrows}} \mathcal{P}\left(\{(x, k_1, v_1, k_2, v_2) \mid x \in \chi,\ k_1 < k_2 \in \iota,\ v_1, v_2 \in \beta\}\right)$$

These constructions easily generalize to arbitrary N indices $k_1, \ldots, k_N$.

Definition 9 (Read, two indices $k_1 < k_2$)**.** The abstraction of $v := a[i]$ is:

$$\begin{aligned} I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, v, i, a_i, k_2, a_{k_2}) \wedge \\ I_1^\sharp(\mathbf{x}, i, v, i, a_i, k_1, a_{k_1}) \wedge i < k_1 < k_2 \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \end{aligned} \tag{37}$$

$$\begin{aligned} I_1^\sharp(\mathbf{x}, i, v, i, a_i, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \wedge \\ I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, a_i) \wedge k_1 < i < k_2 \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \end{aligned} \tag{38}$$

$$\begin{aligned} I_1^\sharp(\mathbf{x}, i, v, k_2, a_{k_2}, i, a_i) \wedge I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, a_i) \wedge \\ I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \wedge k_1 < k_2 < i \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \end{aligned} \tag{39}$$

$$I_1^\sharp(\mathbf{x}, i, v, i, a_i, k_2, a_{k_2}) \wedge i < k_2 \implies I_2^\sharp(\mathbf{x}, i, a_i, i, a_i, k_2, a_{k_2}) \tag{40}$$

$$I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, a_i) \wedge k_1 < i \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, i, a_i) \tag{41}$$

This generalizes to N-ary abstraction by considering all orderings of i inside $k_1 < \cdots < k_N$, and for each ordering taking all sub-orderings of size N.

Definition 10 (Write statement, two indices $k_1 < k_2$)**.** The abstraction of $a[i] := v$ is:

$$I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \wedge i \neq k_1 \wedge i \neq k_2 \implies I_2^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \tag{42}$$

$$I_1^\sharp(\mathbf{x}, i, v, i, a_i, k_2, a_{k_2}) \wedge i < k_2 \implies I_2^\sharp(\mathbf{x}, i, v, i, v, k_2, a_{k_2}) \tag{43}$$

$$I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, a_i) \wedge k_1 < i \implies I_2^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, v) \tag{44}$$

Lemma 5. *The abstract forward semantics of the read statement (Def. 9) and of the write statement(Def. 10) are sound w.r.t the Galois connection.*

Example 7 (Selection sort)**.** *Selection sort finds the least element in $a[d \dots h-1]$ (using Prog. 5 as its inner loop) and swaps it with $a[d]$, then sorts $a[d+1, h-1]$. At the end, $a[d_0 \dots h-1]$ is sorted, where d_0 is the initial value of d.*

```
int d = d0;
while (d < h-1) {
  int p = d, b = a[d], f = b, i = d+1;
  while(i < h) { //find_mini
    if (a[i] < b) {
      b = a[i]; p = i;
    }
    i = i+1;
  }
  a[d] = b; a[p] = f;    //swap
  d = d+1;
}
```

Using the rules for the read (Def. 9) and write (Def. 10) statements, we write the abstract forward semantics of this program as a system of Horn clauses.

We wish to prove that, at the end, $a[d_0, h-1]$ is sorted: at the exit node,

$$\forall d_0 \leq k_1 < k_2 < h,\ a[k_1] \leq a[k_2] \tag{45}$$

This is expressed as the final condition:

$$d_0 \leq k_1 < k_2 < h \wedge exit(d_0, h, k_1, a_{k_1}, k_2, a_{k_2}) \implies a_{k_1} \leq a_{k_2} \tag{46}$$

By running a solver on these clauses, we show that the output of selection sort is truly sorted[16] Let us note that this proof relies on nontrivial invariants:[17]

$$\forall k_1, k_2,\ d_0 \leq k_1 < d \wedge k_1 \leq k_2 < h \implies a[k_1] \leq a[k_2] \tag{47}$$

This invariant can be expressed in our Horn clauses as:

$$d_0 \leq k_1 < d \wedge k_1 < k_2 < h \wedge outerloop(d_0, d, h, k_1, a_{k_1}, k_2, a_{k_2}) \implies a_{k_1} \leq a_{k_2} \tag{48}$$

If this invariant is added to the problem as an additional query to prove, solving time is reduced from 6 min to 1 s. It may seem counter-intuitive that a solver takes less time to solve a problem with an additional constraint; but this constraint expresses an invariant necessary to prove the solution, and thus nudges the solver towards the solution.

Our approach is therefore flexible: if a solver fails to prove the desired property on its own, it is possible to help it by providing partial invariants. This is a less tedious approach than having to provide full invariants at every loop header, as common in assisted Floyd-Hoare proofs.

5 Sets and Multisets

Our abstraction for maps may be used to abstract (multi)sets. Let us see for instance how to abstract the multiset of elements of an array, so as to show that the output of a sorting algorithm is a permutation of the input.

In Ex. 7, we showed how to prove that the output of selection sort is sorted. This is not enough for functional correctness: we also have to prove that the output is a permutation of the input, or, equivalently, that the multiset of elements in the output array is the same as that in the input array.

Let us remark that it is easy to keep track, in an auxiliary map, of the number $\#a(x)$ of elements of value x in the array $a[]$. Only write accesses to $a[]$ have an influence on $\#a$: a write $a[i] := v$ is replaced by a sequence:

$$\#a(a[i]) := \#a(a[i]) - 1;\ a[i] := v;\ \#a(v) := \#a(v) + 1 \tag{49}$$

[16] In Ex. 8 we shall see how to prove that the multiset of elements in the output is the same as in the input.

[17] Nontrivial in the sense that a human user operating a Floyd-Hoare proof assistant typically does not come up with them so easily.

(that is, in addition to the array write, the count of elements for the value that gets overwritten is decremented, and the count of elements for the new value is incremented).

This auxiliary map $\#a$ can itself be abstracted using our approach! Let us now see how to implement this in our abstract forward semantics expressed using Horn clauses. We enrich our Galois connection (Def. 1) as follows:

Definition 11. The *concretization* of $I^\sharp \subseteq \chi \times (\iota \times \beta) \times (\beta \times \mathbb{N})$ is

$$\gamma_\# \left(I^\sharp\right) \;=\; \Big\{(\mathbf{x}, a) \;\mid\; \forall i \in \iota \; \forall v \in \beta \;\; \big(\mathbf{x}, (i, a[i]), (v, \mathrm{card}\{j \in \iota \mid a[j] = v\})\big) \in I^\sharp \Big\} \quad (50)$$

where $\mathrm{card}\, X$ denotes the number of elements in the set X.

The *abstraction* of $I \subseteq \chi \times \mathit{Array}\,(\iota, \beta)$ is

$$\alpha_\# (I) \;=\; \Big\{ \big(\mathbf{x}, (i, a[i]), (v, \mathrm{card}\{j \in \iota \mid a[j] = v\})\big) \;\Big|\; x \in \chi, i \in \iota \Big\} \quad (51)$$

Theorem 6. $\mathcal{P}\,(\chi \times \mathit{Array}\,(\iota, \beta)) \underset{\alpha_\#}{\overset{\gamma_\#}{\leftrightarrows}} \mathcal{P}\,(\chi \times (\iota \times \beta) \times (\beta \times \mathbb{N}))$

The Horn rules for array reads and for scalar operations are the same as those for our first abstraction, except that we carry over the extra two components identically.

Definition 12 (Read statement)**.** Same notations as Def. 2:

$$\begin{aligned} k \neq i \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \wedge \\ I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (k, a_k), (z, a_{\#z})\big) \\ I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (i, a_i), (z, a_{\#z})\big) \end{aligned}$$

Lemma 7. *The abstract forward semantics of the read statement (Def. 12) is a sound abstraction of the concrete semantics given in Def. 2.*

The abstraction of the write statement is more complicated (see the sequence of instructions in Formula 49). To abstract a write $a[i] := v$ between control points p_1 and p_2, we execute a read of the old value from the cell abstraction of the array, decrement the number of cells with this value, execute the write to the cell abstraction, and increment the number of cells with the new value.

Definition 13 (Write statement)**.** With the same notations in Def. 3:

step	translation
`e:=a[i]; #a[e]--; kill(e)`	$a_i \neq z \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$ $I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (a_i, a_{\#z})\big) \wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (a_i, a_{\#z})\big) \implies I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (a_i, a_{\#z} - 1)\big)$
`#a[v]++`	$v \neq z \wedge I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \implies I_b^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$ $I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (v, a_{\#z})\big) \implies I_b^\sharp\big((\mathbf{x}, i, v), (k, a_k), (v, a_{\#z} + 1)\big)$
`a[i]=v`	$i \neq k \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$ $I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (i, v), (z, a_{\#z})\big)$

Lemma 8. *The abstract forward semantics of the write statement (Def. 13) is a sound abstraction of the concrete semantics given in Def. 3.*

If we want to compare the multiset of the contents of an array a at the end of a procedure to its contents at the beginning of the procedure, one needs to keep a copy of the old multiset. It is common that the property sought is a relation between the number of occurrences $\#a(z)$ of an element z in the output array a and its number of occurrences $\#a_0(z)$ in the input array a^0. In the above formulas, one may therefore replace the pair $(z, a_{\#z})$ by $(z, a_{\#z}, a^0_{\#z})$, with $a^0_{\#z}$ always propagated identically.

Example 8. *Consider again selection sort (Program 7). We use the abstract semantics for read (Def. 12) and write (Def. 13), with an additional component $a^0_{\#z}$ for tracking the original number of values z in the array a.*

We specify the final property as the query

$$\mathit{exit}(l_0, h, k, a_k, z, a_{\#z}, a^0_{\#z}) \implies a_{\#z} = a^0_{\#z} \quad (52)$$

Table 1: Comparison on the array benchmarks of [15]. (Average) timing are in seconds, CPU time. Abstraction with $N = 1$. "sat" means the property was proved, "unsat" that it could not be proved. "hints" means that some invariants had to be manually supplied to the solver (e.g. even/odd conditions). A star means that we used another version of the solver. Timeout was 5 mn unless otherwise noted. The machine has 32 i3-3110M cores, 64 GiB RAM, C/C++ solvers were compiled with gcc 4.8.4, the JVM is OpenJDK 1.7.0-85.

Benchmark	Z3/PDR		Z3/Spacer		Eldarica		Comment
	Res	Time	Res	Time	Res	Time	
Correct problems, "sat" expected							
append	sat	2.11	sat	0.85	sat	22.61	
copy	sat	4.66	sat	0.44	timeout(300s)		
find	sat	0.20	sat	0.14	sat	12.93	
findnonnull	sat	0.50	sat	0.34	sat	12.04	
initcte	sat	0.16	sat	0.26	sat	13.28	
init2i	sat	0.31	sat	0.16	sat	14.67	
partialcopy	sat	1.88	sat	0.34	timeout(300s)		
reverse	sat	40.70	sat	2.19	timeout(300s)		
strcpy	sat	0.92	sat	0.37	sat*	66.69	
strlen	sat	0.24	sat	0.22	sat	36.69	
swapncopy	sat	71.16	timeout(300s)		timeout(300s)		
memcpy	sat	3.54	sat	0.39	timeout(300s)		
initeven	sat	1.32	sat	0.71	timeout(300s)		"hints"
mergeinterleave	sat	39.49	sat	4.61	timeout	322.39	"hints"
Incorrect problems, "unsat" expected							
copyodd_buggy	unsat	0.08	unsat	0.04	unsat	7.42	
initeven_buggy	unsat	0.06	unsat	0.06	unsat	6.28	
reverse_buggy	unsat	1.88	unsat	1.28	unsat	58.96	
swapncopy_buggy	unsat	3.13	unsat	0.74	unsat	27.54	
mergeinterleave_buggy	unsat	1.16	unsat	0.56	unsat	31.22	

6 Experiments

Implementation We implemented our prototype VAPHOR in 2k lines of OCAML. VAPHOR takes as input a mini-Java program (a variation of WHILE with array accesses, and assertions) and produces a SMTLIB2 file[18]. The core analyzer implements the translation for one and two-dimensional arrays described in Section 3 and Section 4, and also the direct translation toward a formula with array variables.

Experiments We have tested our analyzer on several examples from the literature, including the array benchmark proposed in [15] also used in [3] (Table 1); and other classical array algorithms including *selection sort*, *bubble sort* and *insertion sort* (Table 2). We compared our approach to existing Horn clause solvers capable of dealing with arrays. All these files are provided as supplementary material.

Limitations Our tool does not currently implement the reasoning over array contents (multiset of values). Experiments for these were thus conducted by manually applying the transformations described in this article in order to obtain a system of Horn clauses. For this reason, because applying rules manually is tedious and error-prone, the only sorting algorithm for which we have checked that the multiset of the output is equal to the multiset of the inputs is selection sort. We are however confident that the two other algorithms would go through, given that they use similar or simpler swapping structures.

Some examples from Dillig, Dillig, and Aiken [15] involve invariants with even/odd constraints. The Horn solvers we tried do not seem to be able to infer invariants involving divisibility predicates unless

[18] http://smtlib.cs.uiowa.edu/

Table 2: Other array-manipulating programs, including various sorting algorithms. a star means that we used another version of the solver, R1 means `random_seed`=1. The ~~striked out~~ result is likely a bug in Z3; the alternative is a bug in Spacer, since the same system cannot be satisfiable and unsatisfiable at the same time.

Benchmark	N	Z3/PDR		Z3/Spacer		Eldarica		Comment
		Res	Time	Res	Time	Res	Time	
bin_search_check	1	sat	0.71	sat	0.34	Crash		
find_mini_check	1	sat	4.22	sat	0.82	sat	110.58	
revrefill1D_check_buggy	1	unsat	0.03	unsat	0.07	unsat	9.21	
array_init_2D	1	sat	0.46	sat	0.22	sat	12.76	
array_sort_2D	1	sat	0.78	sat	0.30	sat	26.68	
selection_sort (sortedness)	2	sat*	99.04	timeout(300s)		timeout(300s)		
selection_sort (sortedness)	2	~~unsat~~	83	sat	48	timeout	334	manual translation
selection_sort (permutation)	1	timeout	600	sat	9.24	timeout	336	manual translation
bubble_sort_simplified	2	sat	5.98	sat	2.77	sat	158.70	
insertion_sort	2	sat(R1)	53.83	timeout(300s)		timeout(300s)		

these predicates were given by the user. For these cases we added these even/odd properties as additional invariants to prove.

Efficiency caveats Our tool does not currently simplify the system of Horn clauses that it produces. We have observed that, in some cases, manually simplifying the clauses (removing useless variables, inlining single antecedents by substitution...) dramatically reduces solving times. Also, precomputing some simple scalar invariants on the Horn clauses (e.g. $0 \leq k < i$ for a loop from k to $i-1$) and asserting them as assertions to prove in the Horn system sometimes reduces solving time.

We have observed that the execution time of a Horn solver may dramatically change depending on minor changes in the input, pseudo-random number generator seed, or version of the solver. For instance, the same version of Z3 solves the same system of Horn clauses (proving the correctness of selection sort) in 3m 40s or 3h 52m depending on whether the random seed is 1 or 0.[19]

Furthermore, we have run into numerous problems with solvers, including one example that, on successive versions of the same solver, produced "sat" then "unknown" and finally "unsat", as well as crashes.

For all these reasons, we believe that solving times should not be regarded too closely. The purpose of our experimental evaluation is not to benchmark solvers relative to each other, but to show that our abstraction, even though it is incomplete, is powerful enough to lead to fully automated proofs of functional correctness of nontrivial array manipulations, including sorting algorithms. Tools for solving Horn clauses are still in their infancy and we thus expect performance and reliability to increase dramatically.

7 Related work

7.1 Cell-based abstractions

Smashing The simplest abstraction for an array is to "smash" all cells into a single one — this amounts to removing the k component from our first Galois connection (Def. 1). The weakness of that approach is that all writes are treated as "may writes" or *weak updates*: $a[i] := x$ adds the value x to the values possibly found in the array a, but there is no way to remove any value from that set. Such an approach thus cannot treat initialization loops (e.g. Program 1) precisely.

Exploding At the other extreme, for an array of statically known finite length N (which is common in embedded safety-critical software), one can distinguish all cells $a[0], \ldots, a[N-1]$ and treat them as

[19] We suspect that different choices in SAT lead to different proofs of unsatisfiability, thus different interpolants and different refinements in the PDR algorithm.

separate variables $a_0, \ldots, a_{N-1}$. This is a good solution when N is small, but a terrible one when N is large: i) many analyses scale poorly with the number of active variables ii) an initialization loop will have to be unrolled N times to show it initializes all cells. Both smashing and exploding have been used with success in the Astrée static analyzer [4, 5].

Slices More sophisticated analyses [12, 17, 19, 32, 33] distinguish *slices* or *segments* in the array; their boundaries depend on the index variables. For instance, in array initialization (Program 1), one slice is the part already initialized (indices $< i$), the other the part yet to be initialized (indices $\geq i$). In the simplest case, each slice is "smashed" into a single value, but more refined analyses express relationships between slices. Since the slices are segments $[a, b]$ of indices, these analyses generalize poorly to multidimensional arrays. Also, there is often a combinatorial explosion in analyzing how array slices may or may not overlap.

Cornish et al. [10] similarly apply a program-to-program translation over the LLVM intermediate representation, followed by a scalar analysis.

To our best knowledge, all these approaches factor through our Galois connections $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$, $\underset{\alpha_{2<}}{\overset{\gamma_{2<}}{\leftrightarrows}}$ or combinations thereof: that is, their abstraction can be expressed as a composition of our abstraction and further abstraction — even though our implementation of the abstract transfer functions is completely different from theirs. Our approach, however, separates the concerns of i) abstracting array problems to array-less problems ii) abstracting the relationships between different cells and indices.

Fluid updates Dillig, Dillig, and Aiken [15] extend the slice approach by introducing "fluid updates" to overcome the dichotomy between strong and weak updates. They specifically exclude sortedness from the kind of properties they can study.

Array removal by program transformation Monniaux and Alberti [31] analyze array programs by transforming them into array-free programs, which are sent to a back-end analyzer. The resulting invariants contain extra index variables, which can be universally quantified away, similar to the Skolem constants of earlier invariant inference approaches [16, 27]. We have explained (p. 3) why these approaches are less precise than ours.

Another difficulty they obviously faced was the limitations of the back-end solvers that they could use. The integer acceleration engine FLATA severely limits the kind of transition relations that can be considered and scales poorly. The abstract interpreter CONCURINTERPROC can infer disjunctive properties (necessary to distinguish two slices in an array) only if given case splits using observer Boolean variables; but the cost increases greatly (exponentially, in the worst case) with the number of such variables.

7.2 Horn clauses

Transformations De Angelis et al. [14] start from a system of Horn clauses over array variables and apply a sequence of transformation rules that i) either yield the empty set of clauses, meaning the program is correct ii) either yield the "false" fact, meaning the program is incorrect iii) either fails to terminate. These rules are based on the axioms of arrays and on a generalization scheme for arithmetic predicates using widening and convex hull.

In contrast to theirs, our approach i) does not require a target property to prove (though the backend solver may need one) ii) does not mix concerns about arrays and arithmetic constraints iii) can prove the correctness of the full insertion sort algorithm (they can prove only the inner loop).

Instantiation Bjørner, McMillan, and Rybalchenko [3] propose an approach for solving universally quantified Horn clauses: a Horn clause $(\forall x\ P(x, y)) \to Q(y)$, not handled by current solvers, is abstracted by $P(x_1(y)) \wedge \cdots \wedge P(x_n(y)) \to Q(y)$ where the x_i are heuristically chosen instantiations. Our approach can be construed as an application of their approach to the axioms of arrays, with specific instantiation heuristics.

We improve on their interesting contribution in several ways. i) Instead of presenting our approach as a heuristic instantiation scheme, we show that it corresponds to specific Galois connections, which clarifies what abstraction is done and what kind of properties can or cannot be represented. ii) We

handle sortedness properties. None of their examples deal with sortedness and it is unclear how their instantiation heuristics would behave on them. iii) We handle multisets (and thus permutation properties) by reduction to arrays. It is possible that our approach in this respect can be described as an instantiation scheme over the axioms for arrays (including the multiset of array contents), but, again, it is unclear how their instantiation heuristics would behave in this respect.

Their approach has not been implemented except in private research prototypes; we could not run a comparison.[20]

7.3 Predicate abstraction, CEGAR and array interpolants

There exist a variety of approaches based on counterexample-guided abstraction refinement using *Craig interpolants* [28, 29, 30]. In a nutshell, Craig interpolants are predicates suitable for proving, using Hoare triples, that some unfolding of the execution cannot lead to an error state. They are typically processed from the proof of unsatisfiability of the unfolding produced by an SMT solver.

Generating good interpolants from purely arithmetic problems is already a difficult problem, and generating good universally quantified interpolants on array properties has proved even more challenging [1, 2, 23].

7.4 Acceleration

Bozga et al. [7] have proposed a method for accelerating certain transition relations involving actions over arrays, outputting the transitive closure in the form of a *counter automaton*. Translating the counter automaton into a first-order formula expressing the array properties however results in a loss of precision.

8 Conclusion and perspectives

We have proposed a generic approach to abstract programs and universal properties over arrays (or arbitrary maps) by syntactic transformation into a system of Horn clauses without arrays, which is then sent to a solver. This transformation is powerful enough to prove, fully automatically and within minutes, that the output of selection sort is sorted and is a permutation of the input.

While some solvers have difficulties with the kind of Horn systems that we generate, some (e.g. SPACER) are capable of solving them quite well. We have used the stock version of the solvers, without tuning or help from their designers, thus higher performance is to be expected in the future. If the solver cannot find the invariants on its own, it can be helped by partial invariants from the user.

As experiments show, our approach significantly improves on the procedures currently in array-capable Horn solvers, as well as earlier approaches for inferring quantified array invariants: they typically cannot prove sorting algorithms.

Our rules are for forward analysis: a solution to our Horn clauses defines a super-set of all states reachable from program initialization, and the desired property is proved if this set is included in the property. We intend to investigate *backward analysis*: find a super-set of the set of all states reachable from a property violation, not intersecting the initial states.

One advantage of some of the approaches (the abstract interpretation ones from Sec. 7.1 and the transformation from [31]) is that they are capable of inferring what a program does, or at least a meaningful abstraction of it (e.g. "at the end of this program all cells in the array a contains 42") as opposed to merely proving a property supplied by the user. Our approach can achieve this as well, *provided it is used with a Horn clause solver that provides interesting solutions without the need of a query*. This Horn clause solver should however be capable of generating disjunctive properties (e.g. $(k < i \wedge a_k = 0) \vee (k \geq i \wedge a_k = 42)$); thus a simple approach by abstract interpretation of the Horn clauses in, say, a sub-class of the convex polyhedra, will not do. We know of no such Horn solver; designing one is a research challenge. Maybe certain partitioning approaches used in sequential program verification [21, 34] may be transposed to Horn clauses.

We have considered simple programs operating over arrays or maps, as opposed to a real-life programming language with *objects*, references or, horror, pointer arithmetic. Yet, our approach can be adapted to such languages, following methods that view memory as arrays [6], whose disjointness is proved by

[20] Their approach is *not* implemented in Z3 (personal communication from N. Bjørner).

typing (e.g. two values of different types can never be aliased, two fields of different types can never be aliased) or by alias analysis.

Acknowledgments We wish to thank the anonymous referees for their careful reading and helpful comments.

References

[1] F. Alberti and D. Monniaux. "Polyhedra to the rescue of array interpolants". In: *Symposium on applied computing (Software Verification & Testing)*. ACM, 2015, pp. 1745–1750. DOI: 10.1145/2695664.2695784.

[2] F. Alberti et al. "An extension of lazy abstraction with interpolation for programs with arrays". In: *Formal Methods in Systems Design* 45.1 (2014), pp. 63–109. DOI: 10.1007/s10703-014-0209-9.

[3] N. Bjørner, K. McMillan, and A. Rybalchenko. "On solving universally quantified Horn clauses". In: *Static Analysis Symposium (SAS)*. 2013, pp. 105–125. DOI: 10.1145/2695664.2695784.

[4] B. Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *Programming language design and implementation (PLDI)*. ACM. 2003, pp. 196–207. DOI: 10.1145/781131.781153. arXiv: cs/0701193.

[5] B. Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software". In: *The Essence of Computation: Complexity, Analysis, Transformation*. LNCS 2566. Springer, 2002, pp. 85–108. DOI: 10.1007/3-540-36377-7_5.

[6] Richard Bornat. "Proving Pointer Programs in Hoare Logic". In: *Mathematics of Program Construction (MPC)*. Ed. by Roland Carl Backhouse and José Nuno Oliveira. Vol. 1837. LNCS. Springer, 2000, pp. 102–126. DOI: 10.1007/10722010_8.

[7] M. Bozga et al. "Automatic Verification of Integer Array Programs". In: *Computer-aided verification (CAV)*. 2009, pp. 157–172. DOI: 10.1007/978-3-642-02658-4_15.

[8] A.R. Bradley, Z. Manna, and H.B. Sipma. "What's Decidable About Arrays?" In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2006, pp. 427–442. DOI: 10.1007/11609773_28.

[9] Jürgen Christ. "Interpolation Modulo Theories". PhD thesis. University of Freiburg, 2015. DOI: 10.6094/UNIFR/10342.

[10] J. Cornish et al. "Analyzing array manipulating programs by program transformation". In: *Logic-Based Program Synthesis and Transformation (LOPSTR)*. Springer, 2014. DOI: 10.1007/978-3-319-17822-6_1.

[11] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". In: *J. Log. Comput.* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[12] P. Cousot, R. Cousot, and F. Logozzo. "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *Principles of Programming Languages (POPL)*. ACM, 2011, pp. 105–118. DOI: 10.1145/1926385.1926399.

[13] Patrick Cousot and Radhia Cousot. "Invited Talk: Higher Order Abstract Interpretation. Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis". In: *IEEE International Conference on Computer Languages*. IEEE, 1994, pp. 95–112.

[14] Emanuele De Angelis et al. "A Rule-based Verification Strategy for Array Manipulating Programs". In: *Fundamenta Informaticae* 140.3–4 (2015), pp. 329–355. DOI: 10.3233/FI-2015-1257.

[15] Işıl Dillig, Thomas Dillig, and Alex Aiken. "Fluid Updates: Beyond Strong vs. Weak Updates". In: *European Conference on Programming Languages and Systems (ESOP)*. 2010, pp. 246–266. DOI: 10.1007/978-3-642-11957-6_14.

[16] C. Flanagan and S. Qadeer. "Predicate abstraction for software verification". In: *POPL*. 2002, pp. 191–202.

[17] D. Gopan, T.W. Reps, and S. Sagiv. "A framework for numeric analysis of array operations". In: *Principles of Programming Languages (POPL)*. 2005, pp. 338–350. DOI: 10.1145/1040305.1040333.

[18] Arie Gurfinkel et al. "The SeaHorn Verification Framework". In: *Computer-aided verification (CAV)*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. LNCS. Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[19] N. Halbwachs and M. Péron. "Discovering properties about arrays in simple programs". In: *Programming language design and implementation (PLDI)*. ACM, 2008, pp. 339–348. DOI: 10.1145/1375581.1375623.

[20] J.Y. Halpern. "Presburger arithmetic with unary predicates is Π_1^1 complete". In: *J. Symbolic Logic* 56.2 (1991), pp. 637–642. ISSN: 0022-4812. DOI: 10.2307/2274706.

[21] J. Henry, D. Monniaux, and M. Moy. "Succinct Representations for Abstract Interpretation". In: *Static Analysis Symposium (SAS)*. 2012, pp. 283–299. DOI: 10.1007/978-3-642-33125-1_20.

[22] K. Hoder and N. Bjørner. "Generalized Property Directed Reachability". In: *Theory and Applications of Satisfiability Testing (SAT)*. Vol. 7317. LNCS. Springer, 2012, pp. 157–171. ISBN: 978-3-642-31611-1. DOI: 10.1007/978-3-642-31612-8_13.

[23] R. Jhala and K.L. McMillan. "Array Abstractions from Proofs". In: *Computer-aided verification (CAV)*. 2007, pp. 193–206. DOI: 10.1007/978-3-540-73368-3_23.

[24] A. Komuravelli, A. Gurfinkel, and S. Chaki. "SMT-Based Model Checking for Recursive Programs". In: *Computer-aided verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. LNCS. Springer, 2014, pp. 17–34. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_2.

[25] A. Komuravelli et al. "Automatic Abstraction in SMT-Based Unbounded Software Model Checking". In: *Computer-aided verification (CAV)*. Vol. 8044. Springer, 2013, pp. 846–862. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_59.

[26] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008. ISBN: 978-3-540-74104-6.

[27] S.K. Lahiri and R.E. Bryant. "Indexed Predicate Discovery for Unbounded System Verification". In: *Computer Aided Verification (CAV)*. 2004, pp. 135–147. DOI: 10.1007/978-3-540-27813-9_11.

[28] Kenneth L. McMillan. "Applications of Craig Interpolation to Model Checking". In: *ICATPN*. Vol. 3536. LNCS. Springer, 2005, pp. 15–16. ISBN: 3-540-26301-2. DOI: 10.1007/11494744_2.

[29] K.L. McMillan. "Interpolants from Z3 proofs". In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 19–27. ISBN: 978-0-9835678-1-3.

[30] K.L. McMillan. "Lazy Abstraction with Interpolants". In: *Computer-aided verification (CAV)*. 2006, pp. 123–136. DOI: 10.1007/11817963_14.

[31] David Monniaux and Francesco Alberti. "A Simple Abstraction of Arrays and Maps by Program Translation". In: *Static Analysis Symposium (SAS)*. Ed. by Sandrine Blazy and Thomas Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 217–234. ISBN: 978-3-662-48288-9. DOI: 10.1007/978-3-662-48288-9_13.

[32] M. Péron. "Contributions to the Static Analysis of Programs Handling Arrays". Theses. Université de Grenoble, Sept. 2010. URL: https://tel.archives-ouvertes.fr/tel-00623697.

[33] V. Perrelle. "Analyse statique de programmes manipulant des tableaux". Theses. Université de Grenoble, Feb. 2013. URL: https://tel.archives-ouvertes.fr/tel-00973892.

[34] X. Rival and L. Mauborgne. "The trace partitioning abstract domain". In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007). DOI: 10.1145/1275497.1275501.

[35] P. Rümmer, H. Hojjat, and V. Kuncak. "Classifying and Solving Horn Clauses for Verification". In: *Verified Software: Theories, Tools, and Experiments (VSTTE) 2013, revised selected papers*. Ed. by Ernie Cohen and Andrey Rybalchenko. Vol. 8164. LNCS. Springer, 2014, pp. 1–21. DOI: 10.1007/978-3-642-54108-7_1.

[36] P. Rümmer, H. Hojjat, and V. Kuncak. "Disjunctive Interpolants for Horn-Clause Verification". In: *Computer-aided verification (CAV)*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Springer, 2013, pp. 347–363. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_24.